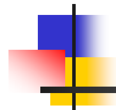
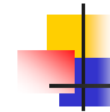


Uke 9 - Repetisjon av metoder, klasser og objekter Innkapsling: private og public Statisk programmering vs. programmering med objeker

18 okt. 2005,

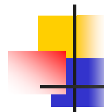
Arild Waaler

Inst. for informatikk, UiO



Hva er en metode

- En metode er en valgfritt antall programsetninger vi gir et navn
- All kode i programmet er inne i en metode (som igjen er inne i en eller annen klasse)
- Skille mellom
  - å *deklarere* en metode (= skrive Javakode for og kompilere)
  - *Utføre* en metode (det som skjer når vi kaller den)
  - Når vi deklarerer en metode, skjer det 'ingen ting '
- En metode blir utført hver gang den kalles fra koden i en annen metode:
  - da hopper utførelsen av programmet til starten av den kalte metoden
  - har den kalt metoden parametere, kopieres verdiene brukt i kallet til metodens parameter-variable (de er som lokale variable i den kalte metoden)



Hva skjer når vi kaller en metode

- Når vi kaller en metode, blir det opprettet et **metodeobjekt** og vi kopierer over verdiene brukt i kallet til parameterne
- Dette metodeobjektet
  - inneholder alle lokale variabler og parameterne til metoden
  - når setningene i metoden utføres, brukes disse variablene og parametrene av metoden
  - metodeobjektet fjernes automatisk når metoden er ferdig utført og returnerer
- Merk forskjellen på å deklarere en metode, og at den utføres.


```
class C {

  int  skrivAntall(int i){
       System.out.println(" Du har kalt meg med:" + i);
       return i+10;
}}

class D
{
  static int dobbel( int k) {
       return 2*k;
  }

  void gjørMye(C cc, int v) {
       System.out.println(" gjørMye kalt");
       int j = cc.skrivAntall(v);
       System.out.println(" 1.verdien av j:" + j);
       j = dobbel(j);
       System.out.println(" 2.verdien av j:" + j);
       System.out.println(" 3.verdien av skrivAntall(j):"
                            + cc.skrivAntall(j) );
  }

  public static void  main ( String[] args)  {
      C c = new C();
      D megSelv  = new D();

      megSelv.gjørMye(c,2);
}}
```

```
>java D
gj°rMye kalt
Du har kalt meg med:2
1.verdien av j:12
2.verdien av j:24
Du har kalt meg med:24
3.verdien av skrivAntall(j):3
```

Hvorfor bruke metoder

- Vi deler opp programmet i metoder fordi:
  - Noen program setninger brukes *flere* steder, eller:
  - Vi vil dele opp programmet i mindre deler
    - Ingen metode bør være lenger enn 30 linjer (og helst mindre)
  - Hver del gjør noe veldefinert som fremgår av navnet:
    - regner ut en bestemt formel
    - skriver ut en meny
    - leser noen data fra terminal eller fil
    - tegner ut opplysninger på skjermen
    - …….

Problemløsning med metoder

- Når vi har laget en metode, og vi har forsikret oss om at den er 'riktig', så har vi laget en *ny operasjon*
- Vi kan nå i resten av koden tenke at vi nå har en slik operasjon tilgjengelig og nytte denne som om den var innebygd i Java
    - eks: skrive ut en meny, regne ut en bestemt formel,..
- Vi trenger da ikke tenke på alle detaljene om *hvordan* denne operasjonen blir utført, bare *at* den blir gjort.
- Vi har da laget et (lite) verktøy som kan gjenbrukes og lettere løse vårt større problem (hele systemet)
- Denne måte å programmere på heter *bottom-up* programmering og nyttes mye.
  - Eks: Java-biblioteket kan best forstås som en diger verktøykasse med nyttige operasjoner og datastrukturer vi kan (og ofte bør) bruke for å lage vårt program

Hva er en klasse

- En klasse er en beskrivelse av hvordan *ett* objekt av en bestemt type i vårt problem er.
  - Inneholder variable som beskriver egenskaper for ett slikt objekt – eks:
    - Navn, adresse, studiepoeng, kurs… for klassen Student
    - Registreringsnummer, eier, type, årsmodel for klassen Bil
  - Inneholder metoder som er fornuftig handlinger for ett slikt objekt – eks:
    - skrivUtVitnemål(), meldPåEmne(),.. i klassen Student
    - beregnÅrsavgift(), skiftEier(),.. i klassen Bil

Skille mellom deklarasjon og bruk av en klasse

- Når vi deklarerer en klasse (= skriver Javakode for) skjer det 'ingen ting' i programmet
- Når vi oversetter og starter opp programmet vårt med javac og java, skjer 'lite':
  - De variable og metodene det står static foran er tilgjengelig
  - Ingen kode (med unntak av main) utføres
- Først når vi sier **`new`** på en klasse, får vi laget et objekt av klassen
  - Objektet inneholder alle variable og metoder som ikke har static foran deklarasjonen (objekt-variable og – metoder)
  - Når vi sier new, kaller vi en konstruktør-metode i klassen, og først når den er ferdig, returnerer new det med det nye objektet

```
class Konto1 {
  String eier;
  int kontoNum, saldo = 0;

  Konto1(String e) {
       eier = e;
  }

  void  settInn(int beløp) {
       saldo = saldo + beløp;
  }

  boolean taUt(int beløp) {
        // moderne bank med muligheter for overtrekk
        saldo = saldo - beløp;
        return saldo > 0;
  }
}

class Bank1
{
  Konto1 [] kontiene = new Konto1[100000];

  public static void  main( String[] args)  {
     Bank1 b = new Bank1();

     for (int i = 0; i < b.kontiene.length; i++) {
        b.kontiene[i] = new Konto1("kunde nr." + i);
        b.kontiene[i].settInn(100);
     }

  }
}
```

Forskjeller mellom klasser og metoder

- Begge lager objekter når de kalles, men:
- Et metode-objekt:
  - fjernes når metoden returnerer
  - inneholder 'bare' variable og parametere som alle er skjult for resten av programmet
- Et objekt laget med **new** fra en klasse:
  - er i hukommelsen etter at det er laget (så lenge det minst er en peker som peker på det)
  - kan inneholde både metoder og variable, som kan nyttes av resten av programmet (med en peker og .)

Ikke alt i et objekt bør være synlig fra resten av programsystemet - innkapsling

- Vi ønsker ofte at resten av systemet bare skal se deler av et objekt
  - eks: **int saldo** i Konto1-objektet bør være skjult, resten av programmet skal bare bruke **settInn() og taUt()** metodene.
- Vi kan regulere tilgangen til variable og metoder ved å sette enten :
  - **private**
  - **public**
  - **protected**
- foran en metode eller deklarasjonen av en variabel

For 'små¨' systemer hvor alle .java filene ligger på samme filområde, gjelder:

- Skriver vi:
  - **ingenting** foran en deklarasjon/metode, så er slike deklarasjoner fullt tilgjengelige for alle annen kode kompilert på samme filområde, men usynlig /sperret for kode kompilert på andre filområder.
  - **private** foran en deklarasjon/metode, så er den bare synlig fra kode i metoder deklarert i *samme klasse*, usynlig/sperret for all annen kode
  - **protected** foran en deklarasjon/metode, så er den synlig i samme klasser og subklasser og synlig i klassene på samme filområdet, men usynlig/sperret i andre klasser (på andre filområder).
  - **public** så er metoden/variabelen synlig for all annen kode.
- Slik delvis sperring av adgang til særlig variable, sikrer oss at vi kan bestemme fullt ut selv i en klasse hvordan en variabel skal endres.

To måter å programmere på

Statisk programmering:

- Var fokus i starten av kurset
- Vi lager ikke objekter av klassene
- Alle variable og metoder er deklarert som static
- Begrepsmessig enkelt, men lite egnet for større programmer

Programmering med objekter:

- Er fokus for resten av kurset (og eksamen).
- Vi lager objekter av klassene (noen eller alle)
- Variable og metoder er vanligvis *ikke* deklarert som static
- Begrepsmessig noe mer komplisert, men mye bedre egnet for større programmer

Objektvariable

Objektvariable er variable på klassenivå som *ikke* er deklarert som static.

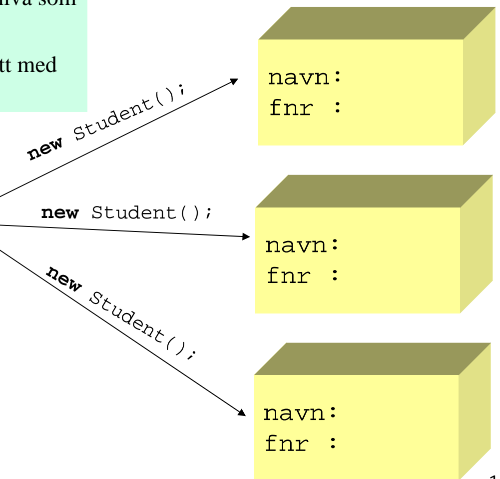
For hvert nytt objekt får vi et fullt sett med nye objektvariable:

```
class Student {
   String navn;
   String fnr;
}
```

new Student();

new Student();

new Student();



Objektvariablenes levetid

```
class Student {
   String navn;
   String fnr;
}
```

Disse variablene blir deklarert når vi lager et objekt av klassen ved å skrive **`new Student()`,** og de lever så lenge objektet lever.

- Objektvariablene blir til når objektet blir til (med new)
- Objektvariablene lever så lenge objektet finnes

Klassevariable

Klassevariable er variable på klassenivå som *er* deklarert som static.

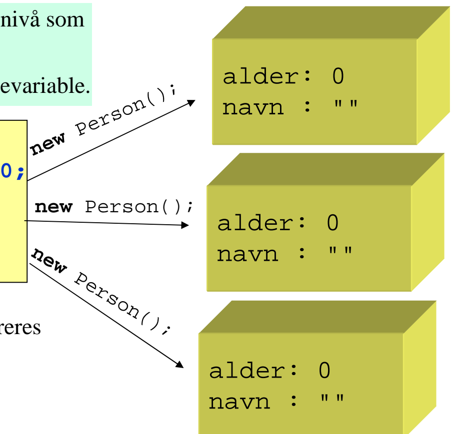
Vi får aldri mer enn ett sett av klassevariable.

```
class Person {
   static int maxAlder=100;
   int alder=0;
   String navn="";
}
```

new Person();

new Person();

new Person();



når klassen refereres første gang

maxAlder: 100

Klassevariablenes levetid

```
class Person {
  static int maxAlder = 100;
  int alder;

  void metode() {...}
}
```

Denne variabelen blir deklarert når klassen Person *blir referert til for første gang* under kjøringen av programmet.

Variabelen lever helt til programmet avsluttes.

Første gang klassen Person blir referert til = første gang programeksekveringen "møter på" Person-klassen, f.eks. :

```
.... new Person() ....
.... Person.maxAlder ....
.... Person.metode() .....
```


Oppsummering: Klassevariable og objektvariable

Objektvariable:

Bare definert i objekter av en klasse.

Hvert objekt har sitt eget sett med objektvariable.

Klassevariable:

Definert selv om det ikke er laget objekter av klassen.

Alle objekter av klassen deler de samme klassevariablene.

Klassemetoder og objektmetoder

Klassemetoder (static-metoder)

- Definert selv om det ikke er laget noen objekter av klassen
- Kan "ses" av alle objekter av klassen
- Kan brukes av andre gjennom dot-notasjon: `<klassenavn>.metode(...)`
- Har ikke tilgang til objektvariable eller objektmetoder

Objektmetoder

- Bare definert i objekter av klassen
- Kan "ses" av objektet som metoden befinner seg i
- Kan brukes av andre gjennom dot-notasjon: `<peker>.metode(...)`
- Har tilgang til alle variable (både klassevariable og objektvariable) og alle metoder (både klassemetoder og objektmetoder)

Statisk programmering

```
class VolumBeregning {
   static double pi = 3.14;
   static int maxAntall = 10;

   public static void main (String [] args) {
     ...
   }

   static double finnVolum(double radius) {
     ...
   }

   static int finnSum(int k) {
     ...
   }
}
```

Alle variable er deklarert som static

Alle metoder er deklarert som static

Programmering med objekter

```
class StudentRegister {
   public static void main (String [] args) {
     ...
   }
}

class Student {
   String navn;
   String fnr;

   void init() {
     ...
   }

   String finnNavn() {
     ...
   }
}
```

Variable er ikke deklarert som static (vanligvis)

Metoder er ikke deklarert som static (vanligvis)

Hvorfor programmere med objekter - 1

- Med objekter kan vi ofte organisere våre data bedre.
  Eksempel:

```
String [] navn = new String[100];
String [] fnr = new String[100];
int [] tlfnr = new int[100];
```

Informasjonen knyttet til en bestemt person er splittet opp i tre arrayer.

```
class Person {
   String navn;
   String fnr;
   int tlfnr;
}

Person [] personreg = new Person[100];
```

Informasjonen knyttet til en bestemt person er samlet i et objekt.

Bedre organisering – særlig når det er mye data å holde orden på.

Hvorfor programmere med objekter - 2

- Med objekter kan vi samle data og operasjoner på dem.

```
... data om studenter ...
... data om ansatte ...
... data om kurs ...
... student-metoder ...
... ansatt-metoder ...
... kurs-metoder ...
```

Her ligger alle data og alle metoder samme sted

```
class Student {
  ... data om studenter ...
  ... student-metoder ...
}
class Ansatt {
   ... data om ansatte ...
   ... ansatt-metoder ...
}
class Kurs {
   ... data om kurs ...
   ... kurs-metoder ...
}
```

Metoder og data som hører sammen er samlet.

Lett å se hvilke metoder som jobber på hvilke data (modularisering av koden).

Lett å kopiere alt som har med personer å gjøre (data+metoder) til andre programmer (gjenbruk).

Hvorfor programmere med objekter - 3

Eksempel: i oblig 3 skal du holde orden på

- en rekke studenter → class Student
- en rekke hybler → class Hybel
- et hybelhus (potensielt flere) → class Hybelhus

En objektorientert løsning (med klassene over) sørger for at

- variabler og metoder som logisk hører sammen ligger også samlet i programkoden
- variabler og metoder som ikke har noe med hverandre å gjøre holdes godt atskilt i programkoden

Analogi: ville du hjemme hos deg selv plassert verktøy, bestikk og CD-plater i samme skuff? Sannsynligheten er stor for at du allerede tenker objektorientert!

Hvordan programmere med objekter?

Grunnregel:
Et objekt skal inneholde data og operasjoner *som naturlig hører sammen.*

Et objekt kan representere:

- et objekt i problemdomenet:
  En konto, en bil, en student, et lån, en eiendom
  (brukes til å definere en datamodell – en "modell av virkeligheten")

- et objekt av mer programteknisk art:
  Et skjermvindu, en fil, en tekststreng, en tabell
  (slike objekter er vanligvis ikke del av datamodellen – det er bare en effektiv måte å gruppere sammen programelementer som hører sammen)

Initialisering av variable i et objekt

Anta at programmet vårt inneholder denne klassen:

```
class Person {
   String navn;
   String fnr;
}
```

Når vi har laget et objekt (med new) ønsker vi normalt å gi variablene i objektet fornuftige verdier med en gang. Noen muligheter:

- **Sett verdiene til objektvariablene direkte med prikk-notasjon:**

```
Person p = new Person();
p.navn = "Petter";
p.fnr = "15108559879";
```

- **Lag en init-metode i klassen:**

```
Person p = new Person();
p.init("Petter", "15108559879");
```

- **Benytt en konstruktør:**

```
Person p = new Person("Petter", "15108535738");
```


Finn- og sett-metoder

Når man skal lage "virkelige" programmer er det vanlig å

- Deklarere alle objektvariable som `private`
- Bruke finn- og sett-metoder for å endre på objektvariable

```
class Eksempel {
  public static void main (String [] args) {
    Student stud = new Student();
    stud.settNavn("Petter");
    System.out.println(stud.finnNavn())
  }
}
class Student {
   private String navn;
   private String fnr;
   void settNavn(String navn) {this.navn = navn;}
   String finnNavn() {return this.navn;}
}
```


Konstruktører - repetisjon

En konstruktør er en spesiell type objektmetode som du kan bruke for å sikre at objektet starter sitt liv med fornuftige verdier i objekt-variablene. Konstruktører

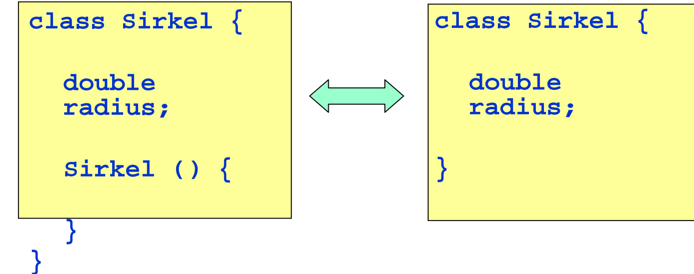
- har alltid samme navn som klassen de ligger i
- utføres automatisk når et objekt opprettes med new
- har ingen returverdi, men skal ikke ha void foran seg
- overlastes ofte, dvs det er ofte flere konstruktører i en og samme klasse, hvor konstruktørene skiller seg fra hverandre ved antall parametre og/eller typen på parametrene.

Eksempel

```
class TestSirkel {
   public static void main (String [] args) {
      Sirkel s1 = new Sirkel();
      System.out.println("Radius: " + s1.radius);
      Sirkel s2 = new Sirkel(5.0);
      System.out.println("Radius: " + s2.radius);
   }
}

class Sirkel {
   double radius;

   Sirkel () {
      radius = 1.0;          } Konstruktør 1
   }

   Sirkel (double r) {
      radius = r;            } Konstruktør 2
   }
}
```


Når det ikke er noen konstruktør

- Når en klasse ikke inneholder noen konstruktør, vil Java selv føye på en "tom konstruktør" uten parametre når programmet kompileres. Dermed er følgende to klassedeklarasjoner ekvivalente:



- Merk: dersom klassen inneholder en eller flere konstruktører, vil Java ikke føye på en "tom konstruktør" uten parametre.